

C++ FRAMEWORK DOCUMENTATION

G53GRA Computer Graphics

Spring Semester 2017/18

This document will provide the necessary details for using the C++ Framework with freeglut to complete your coursework.

Note that the use of this framework is optional and completely at your discretion, however you may find it easier to focus on applying the computer graphics theory with all the windowing handled.

A software document reference for the framework classes is also available [here](#).

Contents

Overview	2
Setting Up	2
Creating your Scene	3
DisplayableObject	4
Virtual Classes	4
Animation	4
Input	5
Example	6
Utilities	6
Texture	6
Camera	7
Demos	8
Appendix	10
Adding Image Resources to a Visual Studio Project	10
Adding Image Resources to an Xcode Project	10
Setting the Platform Toolset for development with Visual Studio 2015	11
Setting the Deployment Target for development with OS X 10.11 or earlier	11
Git Commands	12

Overview

The framework is cross-platform and uses OpenGL, (free)GLUT and GLU. Prebuilt and linked projects are available for use with Windows under Visual Studio and macOS under XCode, containing two folders. The first, "Framework", which contains all the pre-written code for you to use with your project. It is highly recommended that you do NOT edit any of the files in this project: you should just include the header files where appropriate, e.g.:

```
#include "DisplayableObject.h"
#include "Animation.h"
```

The latter folder, named "Code", will contain your coursework code. Some example code, demonstrating various techniques for graphics programming, is provided using the framework, see **Demos**. The main class that you should run will be `MyScene`.
→ `cpp`. There is an overriding method `Initialise()` where you should define all your objects and add them to the scene. Note that you should not attempt to override `Scene::Draw` if you are using this framework.

Setting Up

The framework is hosted in a public repository on the GitHub:

https://github.com/wilocw/g53gra_framework

It is *strongly* recommended you use `git` to clone the repository to your desktop, so you can pull updates and demos directly. If you do not have `git` installed, you can find details for download and installation [here](#) – note: A32 lab machines already have `git` installed.

Run a terminal, or command prompt, and navigate to an appropriate directory location. If you are using a lab machine you will need to navigate to the H: drive in `cmd` first.

```
> H:
> mkdir G53GRA & cd G53GRA
> git clone https://github.com/wilocw/g53gra_framework.git
```

You should **not** be prompted for a username or password. You should see something along the lines of:

```
> git clone
https://github.com/wilocw/g53gra_framework.git
Cloning into 'g53gra_framework'...
remote: Counting objects: 477, done.
```

```
remote: Compressing objects: 100% (282/282), done.
remote: Total 477 (delta 276), reused 348 (delta 181)
Receiving objects: 100% (477/477), 2.74 MiB | 2.61 MiB/s, done.
Resolving deltas: 100% (276/276), done.
Checking connectivity... done.
```

To check this was successful, `cd` into the new folder and check the git status. If the status matches the below, you have cloned the project successfully.

```
> cd g53gra_framework
> git status
On branch coursework
your branch is up-to-date with 'origin/coursework'.
nothing to commit, working directory clean
```

You can now check the project file, for Windows/Visual Studio users it should be located in

```
./G53GRA.Framework/G53GRA.Framework.sln.
```

The XCode project (for macOS users) is found in

```
./G53GRA.Framework/G53GRA.Framework/G53GRA.Framework.xcodeproj
```

The project should build and run without error, and you should see an OpenGL context window open with a black background. This is a success, and everything is set up properly.

Any changes you now make, while writing your coursework, can be added and committed to your local coursework branch to maintain source control over your coursework.

For details on running the demos on the coursework repository, see **Demos**.

For a brief introduction to `git` commands, for use with the framework repository, see **Appendix: Git Commands**.

Creating your Scene

If you have been following the lab tutorials, you should now be familiar with using `freelut`, which is what the framework is built on. The core class is `Scene.cpp`, which handles all the background OpenGL rendering loop and windowing calls. Rather than accessing these functions yourself, the framework has been setup so that you only need to subclass `Scene`, as in `MyScene.h/.cpp`. There are a set of functions that you can override to create your `Scene`, without having to worry about the specific windowing and buffering.

To setup your coursework, you should override the `Initialise()` function. In this function,

you can define instances of `DisplayableObjects` to add to your Scene. These can be added using the `"AddObjectToScene()"` method, which takes a `DisplayableObject*`. These will be rendered (and updated if also subclassing `Animation`) during each rendering loop.

You can set the background colour by calling `glClearColor()` in `Initialise()`, which takes in 4 normalised floats representing RGBA values. Additionally, unless explicitly overridden, the `Initialise()` function will setup default camera and projection settings. It is recommended that you define both of these properties yourself, but you may find it useful when starting out.

Another method you may override includes `Projection()` which handles the projection properties of your Scene. Override this to change the default 'orthographic' settings to 'perspective' mode. It is possible to change title and initial size of your window by changing the input parameters to the `MyScene` constructor.

DisplayableObject

This virtual class contains the requirements for creating objects to display in your Scene: it is contained within the Framework/Interface directory. `DisplayableObject` contains methods for world-space transformations, so you can set (and get) these properties using the `size()`, `position()` and `orientation()` methods, which affect scale, translation and rotation respectively.

Every object that you wish to render in your Scene must publicly subclass `DisplayableObject`. Additional functionalities can be implemented such as animation, user input handling and localised lighting using the provided interfaces.


Virtual Classes

In addition to the `DisplayableObject` virtual class, some additional classes have been provided to give additional, optional functionality to your objects.

Animation

`Animation` is a virtual class containing the function used for animation. It declares the purely virtual method `Update(const double& deltaTime)`, which again must be overridden by all classes that subclass it. Any object with an `Update` function can

be used for animation, which will be given `deltaTime` as a double, representing the runtime since the previous call to `Update()` in seconds.

All objects that subclass `Animation` must be added to the scene as regular `Displayable`  `Objects`, and the `Scene` will differentiate between those that implement `update` and those that don't automatically where appropriate.

Input

If you wish your object to respond to keyboard and mouse input, your object should publicly subclass the `Input` class. This defines methods that will respond to user interaction with the rendering window. Key and mouse status is passed to any objects added to the scene that are defined with public `Input`. The following functions can be implemented in your classes that implement this interface:

`HandleKey(unsigned char key, int state, int mX, int mY)`

Called when ASCII keys are pressed and released, including backspace and Enter/Return (key is a character representing the ASCII key)

`HandleSpecialKey(int key, int state, int mX, int mY)`

Called when special (non-ASCII) keys are pressed and released, such as arrow keys (key is the integer code representing the key)

`HandleMouse(int button, int state, int mX, int mY)`


Called when a mouse button is pressed and released (button is the code for left, right or middle mouse button)

`HandleMouseDown(int mX, int mY)`

Mouse drag, called when mouse is moved while a mouse button is pressed

`HandleMouseMove(int mX, int mY)`

Passive mouse movement, called when no mouse button is pressed

In all functions, the integers, `mX` and `mY`, represent the current mouse position in the screen. This is updated during every input event. For handling key and button presses, the variable `state` simply indicates whether the button has been pressed down (`state`  `== 1`) or released (`state == 0`). This means, for example, every key pressed and released, will call `HandleKey()` twice: once with `state` equal to 1 and again with `state` equal to 0. You should perform checks against this if you only wish code to be executed when a button is pressed but not released!

Example

The following example object details how you may use the virtual classes and `DisplayableObject` to add an object with Input and Animation to your Scene.

```
class Triangle :
    public DisplayableObject,
    public Input,
    public Animation
```

This class definition declares an object `Triangle`, which extends functionality of `DisplayableObject`, `Input` and `Animation`. If an instance is created during `MyScene::Initialise()` and added to the map of objects using `AddObjectToScene()` it will be displayed during each draw session, with calls during `Update()` and when there is keyboard/mouse input. The following override in `Triangle::HandleKey()` will call some defined method `ToggleAnimation()` when the spacebar is pressed (`state == 1`) but not released (otherwise it would be called twice!):

```
void Triangle::HandleKey(unsigned char key, int state, int x,
    int y)
{
    if (state == 1 && key == ' ')
        ToggleAnimation();
}
```

Utilities

Texture

In OpenGL, an image that you load and wish to use as a texture should ideally be a power of 2 in size (this is for caching efficiency). If you have an image that you wish to use you must first ensure it is a power of 2 in size by resizing it in any photo editing software, such as GIMP. To keep the code simple, the image loading capabilities of the C++ framework are limited. *The framework can ONLY load images in bitmap format (.bmp) with either RGB or RGBA colour formats.* The framework will write errors into the debug console if the image loading fails.

To load textures onto the graphics card and onto an OpenGL buffer, the `Scene` class (base class of `MyScene`) provides a function called `Scene::GetTexture(std::`

↪ `string fileName`). This function loads the image at the specified file path and will return an integer corresponding to the buffer number it was loaded into. When you want to use this texture we must pass this buffer number to OpenGL with the call `glBindTexture(buffernumber)`. When we want to turn on texturing we must use the call `glEnable(GL_TEXTURE_2D)` and when we want to stop using texture we disable it with `glDisable(GL_TEXTURE_2D)`. Here is a brief example of using the `GetTexture()` function to load in and use an image as texture:

```
#include "TexturedObj.h"

TexturedObj::TexturedObj (GLuint _texId)
{
    texId = _texId;
}

void TexturedObj::Draw()
{
    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, texId);
    // Draw the object with texture between these calls
    glBindTexture(GL_TEXTURE_2D, 0);
    glDisable(GL_TEXTURE_2D);
}
```

If you are going to have multiple instances of a textured object or multiple objects using the same texture you should only load the texture *once*, otherwise you will fill up all the VRAM on your graphics card and your scene will render very slowly. To do this you should call `Scene::GetTexture()` once in the `MyScene::Initialise()` function and pass the buffer number to each object in their constructor.

As images are loaded from the hard drive, there are a few steps you will need to follow to ensure the image path is relative so that the image can be loaded on every computer you run the coursework on. Please see the appendices for further information about adding image resources with relative paths. See the Appendix for details on how to add images to your Project and use relative paths.

Camera

A `Camera` class has also been provided. This deals with basic functionality for viewing your Scene and navigating using the `wasd`-keys. The initial position of the camera is

at $(0, 0, [h \times \tan 30^\circ] / 2)$ looking towards $(0, 0, 0)$. h represents the window height, and this setup allows you to draw from the origin, rather than translating back in the `Scene` before displaying an object. The y -axis is oriented such that decreasing values of y are down, and increasing values are up.

Implementation has been provided for using the mouse to look around the scene – if the left mouse button is clicked, and the mouse dragged, the view-direction will follow the relative movement of the mouse. Additionally, pressing the spacebar resets the view direction to $(0, 0, -1)$.

Camera also has 2 new function `SetViewport()` and `SetUpCamera()`. `SetViewport` \rightarrow `t()` is called when the user resizes the screen. `SetUpCamera()` is called before each frame is rendered and is used to set the viewing and projection parameters of the scene. The `Camera` class also contains getter function for other classes to be able to query the cameras position or orientation, etc. This will be useful to you if you wish to add billboards or sky-boxes to your scene.

You can edit the code inside of the `Camera` class to add any functionality you wish to use with your coursework, such as adding jumping or crouching. You will also need to change the default projection type from orthographic to perspective.

Demos

To view the demos, you will need to check-out the demo branches using `git`. This can be done in `git bash` or command prompt, to view the triangle example, for example. Run the following checkout command, remember to `cd` into your local `g53gra_framework` directory:

```
> git checkout demo-triangle && git pull
Switched to branch 'demo-triangle'
Your branch is up-to-date with 'origin/demo-triangle'.
Already up-to-date.
```

When reloading Visual Studio, or XCode, you will now see that `MyScene` has been updated. Build and run the project to view the demo. You are free to play around with the code without worrying about messing up your coursework branch. To return to your coursework branch, `commit` or `stash` drop your changes in the demo branch, and run `git checkout coursework` (see **Appendix: Git Commands**).

You can view other demos by running the same `checkout` command for different demo branches, details of which will be made available on Moodle as the course

progresses. Make sure you always run `git pull` to maintain an up-to-date version of the framework. You may also download the `.zip` of the demos directly from the links on Moodle, if you are unfamiliar with git, though it is strongly recommended that you familiarise yourself with source control so you can receive up-to-date demos and any changes made to the framework.

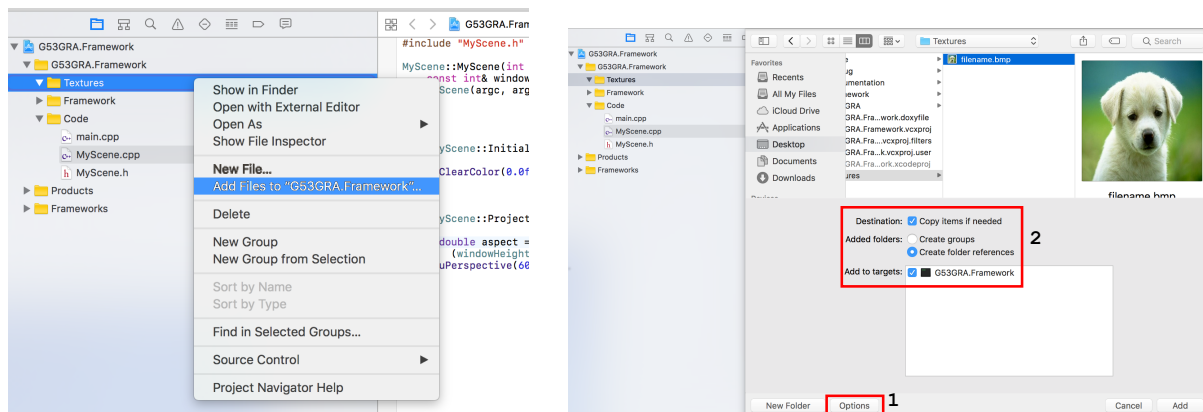
Appendix

Adding Image Resources to a Visual Studio Project

To add images for texturing your project in Windows, simply add the files to the directory `G53GRA.Framework/G53GRA.Framework/Textures`. You can then load them using the **Texture** loader, with path `"./Textures/filename.bmp"`. Remember, only bitmaps (.bmp) are supported.

Adding Image Resources to an Xcode Project

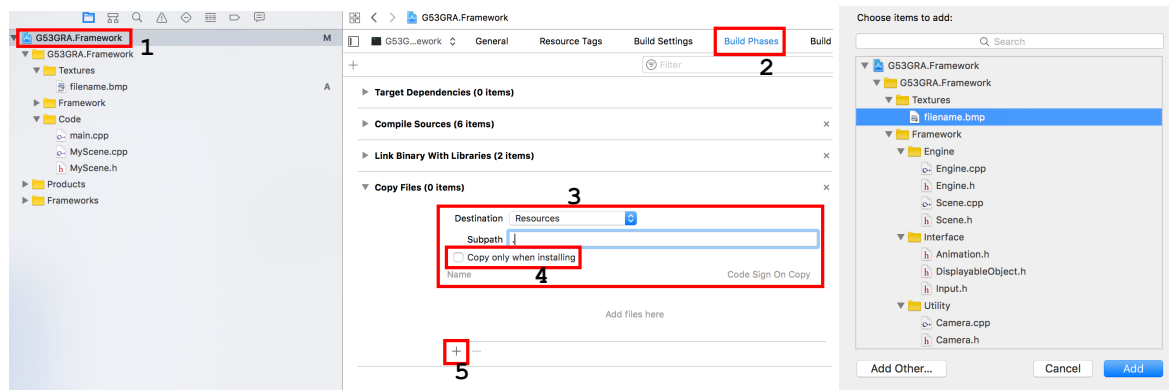
To add images for texturing in an Xcode project, there is an extra step. You should first move your files to the Textures folder, then add the files to the Xcode project. You can do this by right clicking the Textures folder in Xcode, and selecting Add Files to "G53GRA.Framework...". Use the folder navigator to your file, then click "Options" and ensure that you have selected "Copy items if needed" and "Add to targets: G53GRA.Framework", as shown in the screenshot below.



Once the image is included in the project you should add it to the Build Phases, by clicking on the Project name, "G53GRA.Framework" and selecting from the tabs "Build Phases". Under "Copy Files", make sure the Destination drop-down is set to "Resources", and that the Subpath is set to ".". You should also ensure that "Copy only when installing" is *not* ticked.

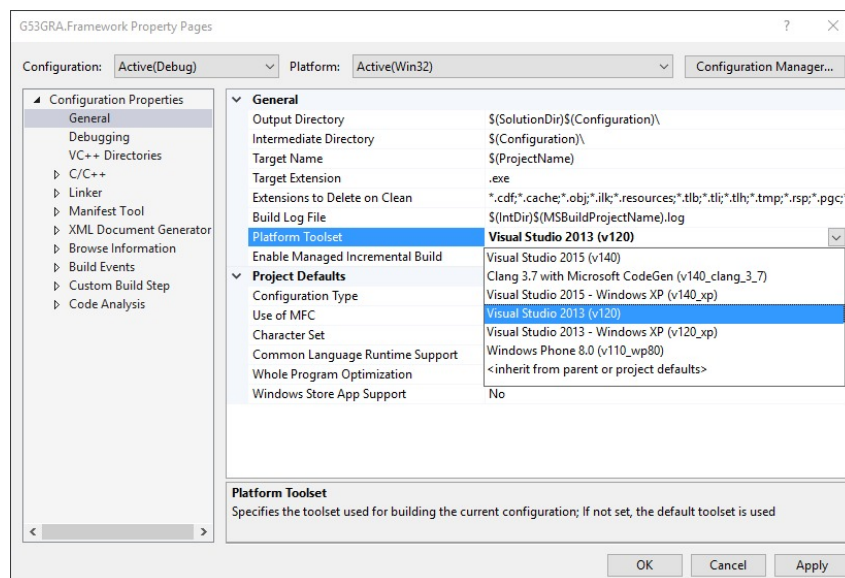
Click the '+' button in the bottom left of the "Copy Files" section, and in the menu that opens, locate and select your image inside the `G53GRA.Framework/Textures` folder. Finally, click "Add" to include your texture file.

You can now load your texture using the **Texture** loader, with path `"./filename.bmp"` → `."`. Remember, only bitmaps (.bmp) are supported.



Setting the Platform Toolset for development with Visual Studio 2015

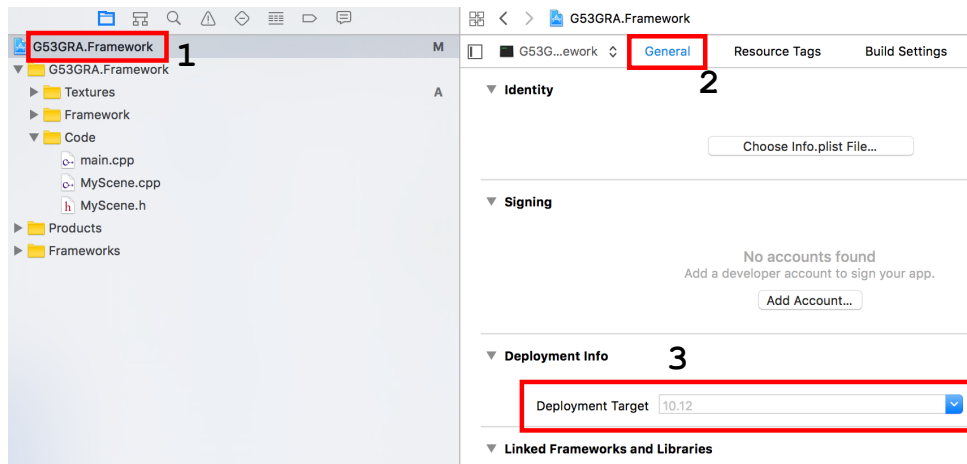
When using Windows, the project framework is setup for Microsoft Visual Studio 2013, in line with the version available in the A32 lab machines. If you are running MSVS 2015, you may get errors when trying to compile. This is due to the Platform Toolset being set to a previous version that you may not have installed. To overcome this error, you should go to "Project > G53GRA.Framework Properties" and under "Configuration Properties > General", set the "Platform Toolset" to your correct version (v140 for Visual Studio 2015).



Setting the Deployment Target for development with OS X 10.11 or earlier

When using macOS, the project framework is setup to build for macOS Sierra (10.12), and you may get an error if you are running an earlier version of the operating system. To change the Deployment Target to your current operating system in Xcode, first click the Project name, "G53GRA.Framework" and select the "General" tab. Under "Deployment

Info", set the Deployment Target drop-down from 10.12 to your running OS version.



Git Commands

Saving your changes

Once you have made changes, and you want to save them or checkout another branch, you will have to `commit` your changes to the current branch (this should normally be coursework).

By running `git add .` followed by `git commit`, you will "add" all your changes to be marked for committing, then add those to your branch as a single revertible "save point". You will need to write a commit message, so the chain of commands will be as follows:

```
> git add .
> git commit -m "commit message"
[coursework #####] commit message
X files changed, Y insertion(+), Z deletion(-)
...
```

The contents of "commit message" should reflect the changes you have made, such that you can easily understand it as a summary of the changes between your commits.

Changing branches

To change branch, normally to view a demo, you need to "checkout" that branch. You can view the list of branches by running `git branch -all`, which will give a list of local branches, and those on the remote server, suffixed "remotes/origin/".

```
> git branch --all
* coursework
remotes/origin/HEAD -> origin/coursework
```

```
remotes/origin/coursework
remotes/origin/demo-triangle
remotes/origin/demo-triforce
remotes/origin/hotfixes
remotes/origin/master
```

You can checkout any of these branches, e.g. `git checkout demo-triangle`. You should always call `git pull` when checking out a new branch. This will download any updates that have been uploaded to the repository, though you may be prompted for your IS username and password.

Troubleshooting

To check the status of the repository, you should run `git status`. If status is fine, and the branch is up-to-date and all changes are committed, you will see the following message:

```
> git status
On branch coursework
Your branch is up-to-date with 'origin/coursework'.
nothing to commit, working tree clean
```

This allows you to checkout other branches. Sometimes, if you are checking out to another branch, it may fail. This is often because you have made changes, or not cleaned the project files, but you may wish to discard these. If they are intentional changes, you should add and commit, as described above.

To review uncommitted or untracked changes, you can use `git status`. The following example arises when the Visual Studio build has not been cleaned:

```
> git status
On branch coursework
Your branch is up-to-date with 'origin/coursework'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   G53GRA.Framework/G53GRA.Framework/Debug/vc120.idb

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        G53GRA.Framework/G53GRA.Framework.VC.opendb

no changes added to commit (use "git add" and/or "git commit -a")
```

One option here is to simply clean the project, but you may also use `git` to drop the

changes. By executing the following command line, you can add the changes to git, add them to a stash and then delete the stash, effectively removing the changes (be careful not to do this with intentional changes!)

```
> git add . && git stash && git stash drop
```

Once you have executed this command, rerun the `git status` command to make sure you now have a clean branch. You can now checkout to another branch if needed.